CONDEnSe: Contract-Based Design Synthesis

César Augusto Santos
Siemens Industry Software NV
KU Leuven
Leuven, Belgium
cesar.augusto@siemens.com

Amr Hany Saleh
Siemens Industry Software NV
Leuven, Belgium
amr.saleh@siemens.com

Tom Schrijvers
KU Leuven
Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Mike Nicolai
Siemens Industry Software NV
Leuven, Belgium
mike.nicolai@siemens.com

***Abstract*—It is difficult to maintain consistency between artifacts produced during the development of mechatronic systems, and to ensure the successful integration of independently developed parts. The difficulty stems from the complex, multidisciplinary nature of the problem, with multiple artifacts produced by each engineering domain, throughout the design process, and across supplier chains.**

In this work, we develop a methodology and a tool, CONDEnSe, that given a set of Assume/Guarantee (A/G) contracts that capture the system requirements, and a high-level decomposition of the system model, automatically generates design variants that respect the requirements and exports those variants to different engineering tools for analysis.

Our methodology makes use of a contract-based design algebra to ensure that generated artifacts for all design variants are consistent by construction, even when the process is modularized and independently developed parts are only later integrated.

In contrast with earlier work, our approach reduces the search space to models that comply with the captured design requirements.

***Index Terms*—contracts, synthesis, mbse, integration**

I. Introduction

Systems engineering [1] is a complex, multidisciplinary field that presents many consistency and integration challenges [2]. Engineers from different fields work on different parts of the system at different levels of abstraction. Furthermore, each engineering domain has its own tools and methodologies, making coordination and integration difficult [3].

Much of the difficulty stems from the multitude of artifacts produced during the design process in incompatible tools with different semantics, from requirement specifications to detailed physical models, and the need to keep them consistent. Inconsistencies cause issues throughout the design and lead to expensive redesign cycles. Consistency is typically managed through traceability links between artifacts [4]. While traceability links cannot by themselves detect consistency issues, they can be used to inform the relevant parties when a related artifact is modified. This involves manual maintenance and is error-prone.

We consider the notion of traceability from Model-Based Systems Engineering (MBSE) [5] where every domain model is linked to a system model [6] that serves as a central source of truth. We aim to improve upon the state-of-the-art in MBSE with a novel design methodology focused on automation. The goal is to expand the set of artifacts that are automatically and fully derived from the system model, thereby reducing the chance for human error and the likelihood of consistency issues.

The work was supported by the Belgian funding agency VLAIO.

We accomplish this through the combination of a synthesis methodology with a formal verification methodology, where the system requirements participate in the synthesis process. Only architectures that follow the system requirements are generated, which can then be exported to various modeling and simulation tools. Our specific contributions are:

- We present a methodology that combines Contract-Based Design [7] with Computational Design Synthesis [8] in a synergistic way where the weaknesses of either approach are reduced (Section III).
- We expand the state of the art in consistency management by expanding the set of artifacts in the design process that can be made consistent by construction, through automatic generation of configurations from the system model, exportable to various tools.
- We demonstrate our methodology with a proof-of-concept tool, CONDEnSe, which we validate by applying it to two use cases (Section IV).

First, we motivate this work through an example of the design process (Section II-A) and the challenges therein, followed by overviews of the capabilities and weaknesses of CDS (Section II-C) and CBD (Section II-D).

II. Motivation

This section presents an example of the design process, the core issues we are interested in, and our solution.

A. Design Process

Consider an automotive Original Equipment Manufacturer (OEM) who follows the V-Model development process [3] to develop a new state-of-the-art hybrid car.

1) First, the system requirements are collected, identifying what is expected of the system. For example, a requirement might be: *the car must be powered by both gasoline and electricity*.
2) From the requirements, a set of functions are derived, e.g., *converting chemical energy into torque*, *converting electric power into torque* and *storing electric power*.
3) From the functions, a set of components that comprise the system is derived. In this case, the system would

need to have a *combustion engine*, an *electric motor* and a *battery* at the minimum. This process often culminates in a Systems Modeling Language (SysML) [6] system model.

4) From the set of components, various logical architectures can be derived. *Series*, *Discontinuous Variable Transmission (DVT)* and *Continuous Variable Transmission (CVT)* architectures are all available on the market [9]. Orthogonally, cars can also be *front-wheel*, *rear-wheel* or *all-wheel* driven, giving at least 9 unique configurations.
5) From the logical architectures, multiple detailed models of the system and its components are produced manually by the engineers: requirement documents, fault tree analysis [10], Computer Assisted Design (CAD) [11] models, 1D [12] and 3D simulations [13], embedded controllers, tests, etc. [6]

If a problem is detected during the evaluation of a detailed model that points to a conflict in the requirements, artifacts related to those requirements, and their derived artifacts, have to be re-evaluated or redone. Design alternatives have to be reconsidered, components have to be adapted, and the respective suppliers must be informed of the necessary changes.

In a real system, an extremely large number of artifacts could be considered during this process. This introduces two challenges:

- How can consistency of these artifacts be maintained across changes in the design?
- How can suppliers develop their parts independently, with minimal coordination, while ensuring safe integration into the complete system?

B. Model Based Systems Engineering (MBSE)

Tackling these challenges is the goal of Model-Based Systems Engineering (MBSE) [5], which currently receives much attention from the industry [14]. MBSE is a paradigm that focuses on domain models as the primary means of information exchange between engineers. More concretely, there is a *system model* from which all other models are derived [6]. Since all models derive from a single source of truth, they *should* be consistent.

However, MBSE does not dictate the structure of the system model nor how the other models derive from it. Many methodologies, tools, and standards have been developed to enable MBSE [5], [6].

C. Computation Design Synthesis (CDS)

Of these methodologies, Computational Design Synthesis (CDS) is of particular interest to us [8]. In CDS, various methods are used to automate the generation of a range of alternatives. These methods may be fully automated or interactive, depending on what is to be synthesized. CDS is a broader concept than optimization, as the range of alternatives need not be optimal; selection of the best alternative is a separate process.

We focus on CDS because the synthesized alternatives are consistent with each other by construction. No maintenance effort is required from the engineer for those artifacts, eliminating the possibility for human error.

In the work of Nicolai et al. [15] a CDS methodology and tool is introduced which generates potentially many logical architectures from a single, SysML-like system model. The components and parameters comprising this system model optionally map to library components in various simulation and analysis tools, with architectural models for those tools then automatically generated. The work of Nicolai et al. was successfully applied to several use cases, from transmission design [16] and hybrid vehicles [17] to safety in avionics [18].

The models generated by the tool of Nicolai et al. are consistent by construction, as changes to the system model are automatically reflected in the tool-specific models. This covers a large portion of the artifacts produced in the design process, from the end of the functional analysis to the physical modeling.

We identified three challenges to this CDS approach:

- How to cover more of the design process within this methodology, particularly requirements?
- How to avoid a combinatorial explosion in the number of architectures generated?
- How to safely modularize the process?

The first two challenges are related. Since requirements do not participate in the synthesis process, nonsensical solutions form the vast majority of the output. For example, an architecture might have the engine directly connected to the wheels of the car without going through the transmission. To prevent the generation of unreasonable solutions, the engineer must add further design knowledge to the model (e.g., the engine may not connect directly to the wheels). However, if too much design knowledge is added, solutions of interest could be filtered out unintentionally, reducing to a single logical architecture in the worst case (i.e., design fixation).

The solution is to capture the requirements within the system model, since this not only increases the number of artifacts that are consistent by construction, but also dramatically improves the quality of the generated output, as only variants that respect the requirements are generated, without accidentally eliminating possible variants due to restrictive manual constraints.

As stated at the start of this section, requirement gathering and analysis is the first phase of the design process. Requirements are typically written in natural language and are a common source of consistency issues [19]. Various methods exist to manage requirements in a less ad-hoc manner [20], [21], but we solely concern ourselves with approaches based on formal methods, as they can contribute to the synthesis process.

D. Contract-Based Design (CBD)

Contract-Based Design (CBD) is one such methodology. Originally intended for software engineering, it was adapted in

various ways to systems engineering, culminating in a meta-theory by Albert Benveniste et al. [7]. In CBD, requirements are organized in modular units called *Contracts*, with the assertions in the contract written in an appropriate, computationally verifiable logic.

CBD mainly focuses on modularity and integration through semantic composability and refinement. It answers the question: how are consistency and integration handled with suppliers (or even different teams in the same organization). Contracts can be decomposed and given to different teams to implement, with the assurance that whatever implementation is produced fits the overall system [7].

We identified two challenges to this approach:

- How to ensure the quality of the contracts (e.g. do they sufficiently restrict the design space)?
- How to capture a wide variety of functional and non-functional requirement specifications that are used in industry?

The first challenge is of particular interest to this paper, and is a general issue with formal methods. A contract that assumes nothing and ensures nothing allows all implementations and gives no assurances. Empty contracts are easily avoidable, but overly lenient contracts present the same issue. We ran afoul of this problem in an avionics use case, which we cover in detail in Section IV-B. Note that, detecting consistency issues between the contracts is not sufficient since an empty contract is trivially consistent.

We claim that CDS and CBD have a mutually beneficial relationship. We originally considered CBD solely as a way to solve the challenges of CDS, but it turned out that CDS also solved the contract quality problem of CBD (see Section III-C). The synthesis process serves as as a sanity check, similar to mutation testing [22], but where the evaluation is manual and reliant on human intuition. If nonsensical architectures are synthesized from the contracts, that means the contracts are overly lenient. We assume engineers use CONDEnSe in an interactive and iterative manner, tweaking the requirements as appropriate until all generated outputs appear sensible from a domain point of view.

On the other hand, with sufficiently strict contracts, the CDS challenges are solved. Requirements are now connected to the system model, participate in the synthesis process (avoiding the explosion of results), and design fixation is minimized as the engineer is no longer required to restrict the model manually.

Table I gives an overview of the consistency assurances and automation our methodology is able to provide for each design phase. The Derivation column states how artifacts are derived from the previous artifacts (e.g. Contracts are derived manually from Textual Requirements). The Consistency column states if the tool assures consistency between artifacts of that type. Partial means that automation and consistency is only possible for a subset of the artifacts.

Design Phase	**Artifacts**	**Derivation**	**Consistency**
Requirements	Textual Requirements	N/A	Manual
	Contracts	Manual	Automatic
Functional	Functions	Manual	Manual
	Components	Manual	Automatic
Logical	System Model	Manual	Automatic
	Architectures	Automatic	Automatic
Physical	Tool-Specific Models	Partial	Partial

TABLE I
CONSISTENCY GUARANTEES OF THE CONTRACT-BASED DESIGN SYNTHESIS APPROACH.

III. METHODOLOGY

In this section we detail how our approach works, first by introducing the Computational Design Synthesis approach, then the Contract-Based Design approach, and finally how they are integrated.

A. Computational Design Synthesis (CDS)

We follow the work of Nicolai et al. [15] for our CDS approach (our solver uses the same encoding, and our language is heavily inspired by their work). The goal is to compute all possible logical architectures (or configurations) that follow a given system model. We model the system in a textual, SysML-inspired Domain-Specific Language (DSL). Other artifacts can be derived from the logical architectures if there appropriate library components for each abstract block. We will exemplify our approach with a Wheel Braking System (WBS) use case, adapted and simplified from [23]. Figures 1 and 2 are textual and graphical representations of the WBS model in our DSL (the representations are isomorphic to each other).

We model the system as a hierarchical decomposition of components. The main component is the Wheel Braking System itself (called `wbs`) and is marked as `system` rather than `component`. The `owns` keyword is used to declare that `wbs` is composed of one `bscu` (the Braking System Control Unit) and one `hydraulics` (an abstraction of the complex machinery that triggers the breaks). The interface of `wbs` is modeled in a `ports` block, where each port consists of a direction (`in`|`out`|`io`) followed by the type of data flowing through that port (a `real` number representing response time in this case), and a name. A port may be marked as optional by surrounding it with parentheses (dashed outline in the graphical representation). In the case of the `wbs`, it has a pedal with a redundant signal forming the two inputs and an output port for the brakes.

The `bscu` is modeled as a subsystem, that is, also as a decomposition of components. Its interface is similar to the `wbs`, with a redundant pedal signal, but its output port is just a command signal rather than a direct trigger for the brakes. It is made out of two `sub_bscu` components and a `switch` (note the multiplicities on the left). The idea is that the `bscu` is a fault-tolerant system made out of non-fault-tolerant parts, wired in such a way that fault-tolerance is achieved.

```
system wbs:
    owns:
        [1] bscu
        [1] hydraulics
    ports:
        in::real pedal_1
        in::real pedal_2
        out::real brake

component bscu:
    owns:
        [2] sub_bscu
        [1] switch
    ports:
        in::real pedal_1
        in::real pedal_2
        out::real cmd

component hydraulics:
    ports:
        in::real cmd
        out::real break

component sub_bscu:
    ports:
        in::real pedal
        out::real cmd
        (out)::bool error

component switch:
    ports:
        in::real cmd_1
        in::real cmd_2
        in::bool error
        out::real cmd
```

Fig. 1. Textual representation of the Wheel Braking System.

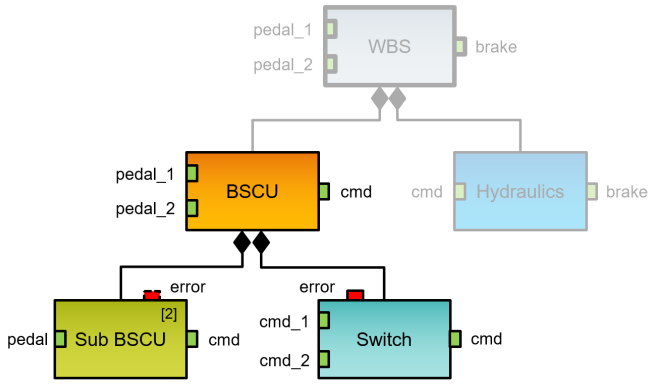


Fig. 2. Graphical representation of the Wheel Braking System. Our focus is on the `bscu`.

We will ignore the `wbs` and the `hydraulics` and focus on the `bscu` and its contents, as the higher level system is of little interest, synthesis-wise.

The goal here is to synthesize possible implementations for the `bscu`, built from two `sub_bscu` parts and a `switch`. The `sub_bscu` components, unlike the parent `bscu`, do not accept redundant pedal signals. If a `sub_bscu` detects a problem in its pedal input, it sends an error signal through its error output port. That port is marked as optional (direction surrounded by parentheses), which means that the port may

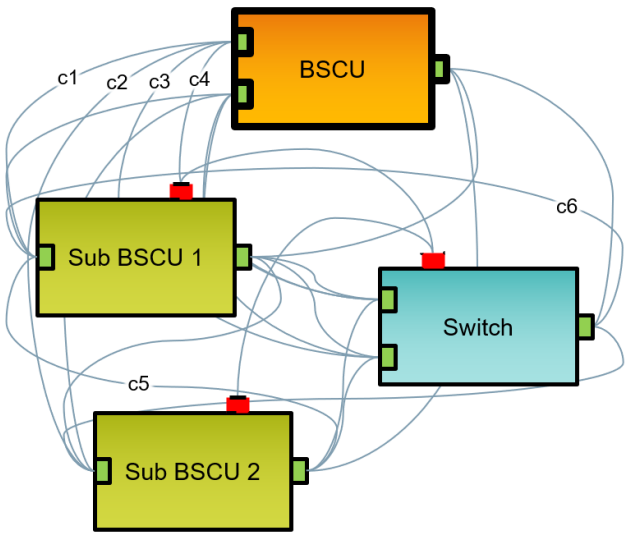


Fig. 3. The web of possible connections in the `bscu`. The connections mentioned in the paper are marked in the picture.

or may not be present in an architecture. We required that all ports must be connected, or forwarded to higher level ports, in the complete system. By making a port optional, we are saying that if it is not connected, then it can be removed from the system. We do this because as the `switch` only has one input error port one of the `sub_bscu` components will not connect to it, though either could.

We proceed as follows to encode the synthesis problem:

- All the ports from all the components we want to synthesize architectures from are collected. In this case, the ports of the `bscu` and all the ports of its sub-components.
- For each non-optional port, all the ports (including optionals) it can connect or map to, are collected. For example, the input of a `sub_bscu` can be mapped to one of the inputs of the `bscu` since it is a higher level component. Its output port can connect to an input port of the switch since it is of the same level.
- For each pair of ports, i.e. a possible connection, a new boolean variable is created. The order in the connection does not matter. Port P1 connected to port P2 is the same as port P2 connected to port P1, only one variable is created.
- For each port, the set of possible connections (the boolean variables previously created) that include it is collected, and a `one-hot` constraint is asserted, i.e. exactly one out of n variables may be true, the rest must be false.

To illustrate the process, the complete web of possible connections is represented graphically in Figure 3. The following are the connections generated for the `pedal_1` input of the `bscu`:

```
c1: bscu.pedal_1 -- sub_bscu.1.pedal
c2: bscu.pedal_1 -- sub_bscu.2.pedal
c3: bscu.pedal_1 -- switch.cmd_1
c4: bscu.pedal_1 -- switch.cmd_2
```

Below are the connections generated for the pedal input of the first instance of a sub_bscu (sub_bscu.1). Note that c1 is reused:

```
c1: bscu.pedal_1 -- sub_bscu.1.pedal
c5: sub_bscu.1.pedal -- sub_bscu.2.cmd
c6: sub_bscu.1.pedal -- switch.cmd
```

Additionally, these are the one-hot constraints generated for both. Observe the shared connection variable:

```
one-hot(c1,c2,c3,c4)
one-hot(c1,c5,c6)
```

The same process is performed for every port in the system. Note that more connections will appear in multiple one-hot constraints. For example, c3 is one of the possible connections for switch.cmd_1.

We need to find an assignment to the connection variables that satisfies all of the one-hot constraints. Solving this constraint problem gives a set of connections such that the system is fully connected and every non-optional port is connected.

We use a SAT [24] solver to solve this constraint problem. Each possible connection is modeled as a separate variable, and a solution is an assignment to these variables. Each one-hot constraint is encoded as a combination of an at-least-one constraint (i.e. a disjunction) with an at-most-one constraint. There are various ways of encoding at-most-one constraints with different performance characteristics [25], but the simplest way is a conjunction of every pair of ports such that at least one port in the pair is false.

Figure 4(a) is one possible solution, or logical architecture for the bscu. Figures 4(b) and 4(c) are other possible architectures. There are in total 68 possible solutions for this problem.

Of the three solutions in Figure 4, only solution (a) shows a valid implementation for the bscu. Solution (b) appears feasible but as we will discuss later, it is not (see Section III-B). Solution (c) is broken as the switch is being bypassed.

We consider both solution (a) and solution (b) to be configurations *of interest*. Configurations of interest are configurations that engineers, given their domain knowledge and experience, but without taking into account requirements or detailed behavioral models, are able to intuitively identify as reasonable structures for their particular domain. A solution like C where the switch is bypassed is clearly incorrect, and therefore is not of interest. Out of the 68 possible solutions, only 8 are considered of interest, all other solutions will result in easily identifiable non-functioning systems (Note that this does not mean all 8 solutions will work in practice).

This happens because we have not added sufficient domain knowledge to the system model. A simple improvement would be to add a constraint that the output of the switch must be the output of the bscu (e.g. through an *enforced connection* constraint between bscu.cmd and switch.cmd). This would eliminate all solutions that completely bypass the switch, but it still results in invalid solutions.

For a simple example, this is not a problem; 68 solutions can be checked manually, or simulated in a small amount of time. For larger systems however, with thousands of possible configurations, added design knowledge is paramount, but must be carefully handled as it may lead to design fixation.

In our system, we use Contract-Based Design (CBD) [7] to enable requirements to serve as domain knowledge and to filter out non-complaint solutions.

B. Contract-Based Design (CBD)

We previously mentioned that only 8 out of 68 solutions are sound from an engineering point of view. However, these solutions might still not be sound with respect to the system requirements. We are interested in producing solutions that are not only of interest, but that also observe the system requirements.

The system requirements for the bscu are as follows, adapted and simplified from [23]:

1) R1: Either one of the inputs to the bscu must arrive within 4 milliseconds of the user triggering the brakes.
2) R2: The bscu must fire a signal on its cmd port after receiving a signal on either input port, within 6 milliseconds total.
3) R3: The sub_bscu must fire a signal on its cmd port 1 millisecond after receiving an input if that input arrives within 4 milliseconds, otherwise it must signal an error.
4) R4: The switch must select the input from its cmd_1 port if there is no error signal, otherwise it must select the input from its cmd_2 port, and pass the signal within 1 millisecond.

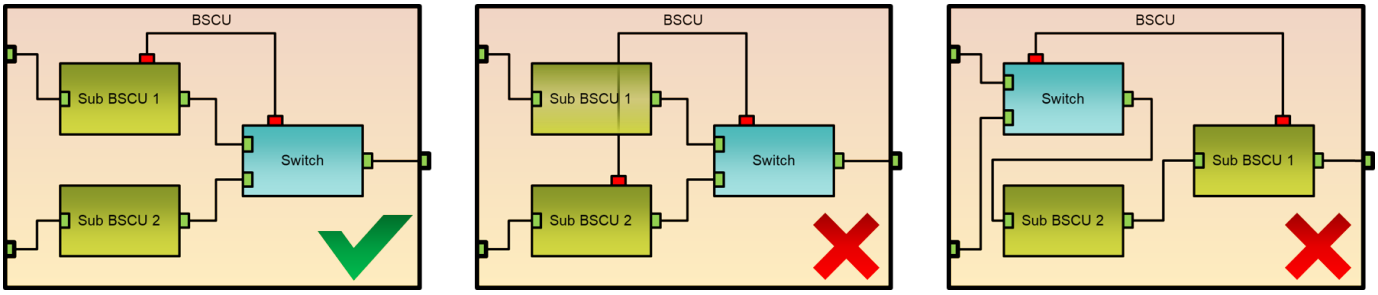
To enable these requirements to participate in the synthesis process, we use a contract-based design approach. The process to convert requirements into contracts is as follows:

1) Separate the requirements per component, splitting the requirements if necessary. In this case, R1 and R2 belong to the bscu, R3 to the sub_bscu and R4 to the switch.
2) Identify which requirements are *assumptions* about the environment, and which are *guarantees* the component must provide. In this case, R1 is an assumption, while R2, R3 and R4 are guarantees.
3) Encode the assumptions and guarantees in an appropriate logic. In this case, a combination of propositional logic with real arithmetic is sufficient.

Through this process we arrive at the contracts in Figure 5. Given these contracts, we want to find out for a particular system configuration if the *composition* of the sub_bscu components and the switch is a valid *refinement* of the bscu.

The operations of *composition* ($\otimes$) and *refinement* ($\preceq$) are defined in the contract meta-theory by Benveniste et al. [7]. These meta-operations can be defined for various sorts of contracts, but for our use case we are only concerned with their definition for Assume/Guarantee (A/G) contracts.

The refinement operation ($\preceq$) is defined as follows, where A, G are the assumptions and guarantees of C (the top level



(a) (b) (c)

Fig. 4. Three `bscu` implementations. Solution (b) is just a small change from Solution (a), but it does not follow the system requirements. The switch always picks the second input when it detects an error, so it will pass along the erroneous command from `sub_bscu.2` in this case.

```
contract bscu:
    ports:
        in::real pedal_1
        in::real pedal_2
        out::real cmd
    assumptions: # R1
        pedal_1 < 4 or pedal_2 < 4
    guarantees:  # R2
        cmd >= min(pedal_1, pedal_2)
        cmd <= 6.0

contract sub_bscu:
    ports:
        in::real pedal
        out::real cmd
        (out)::bool error
    guarantees:  # R3
        if pedal < 4
            then cmd == pedal + 1
            else error

contract switch:
    ports:
        in::real cmd_1
        in::real cmd_2
        in::bool error
        out::real cmd
    guarantees:  # R4
        if not error
            then cmd == cmd_1 + 1
            else cmd == cmd_2 + 1
```

Fig. 5. Contracts for the various components in the BSCU.

contract) and A', G' are the assumptions and guarantees of C' (the refinement contract):

$$C' \preceq C \text{ if } A \Rightarrow A' \wedge G' \Rightarrow G \tag{1}$$

This formula says that the assumptions of the `bscu` must be equal to or stronger than the assumptions of its refinement, and the guarantees of the `bscu` must be equal to or weaker than the guarantees of its refinement. More concretely, it must be possible to use the refined implementation anywhere the original system would be used, meaning fewer (or equal) assumptions about the surrounding environment, and it must accomplish the same tasks as well or better, meaning more (or equal) guarantees. In our case, the refinement of the `bscu` is a composition of three components (two `sub_bscu` instances and one `switch`).

The composition operation for Assume/Guarantee definition is defined as follows, where C is the resulting composition contract, C_1 and C_2 are the contracts of the components being composed, and A, G, A_1, G_1, A_2, G_2 are the corresponding assumptions and guarantees:

$$\begin{aligned} C &= C_1 \otimes C_2 \\ G &= G_1 \wedge G_2 \\ A &= \max \left\{ A \;\middle|\; \begin{matrix} A \wedge G_2 \Rightarrow A_1 \\ A \wedge G_1 \Rightarrow A_2 \end{matrix} \right. \end{aligned} \tag{2}$$

What this formula says is that the guarantees of a composition are the conjunction of the guarantees of the constituent components, and that if the guarantees of one component partially fulfill the assumptions of the other, the assumptions of the composition are reduced accordingly.

This formula does not lend itself to efficient implementation, but it is not the only way of modeling the composition operation. We first transform the contracts into a *saturated form* as in [7], where the assumptions imply the guarantee. This transformation can always be done:

$$\begin{aligned} A_{saturated} &= A \\ G_{saturated} &= A \Rightarrow G \end{aligned} \tag{3}$$

From this point forward, A and G refer to their saturated forms. With a saturated contract, we can use the following formula for composition, where A, G, A_1, G_1, A_2, G_2 are the assumptions and guarantees of each respective contract:

$$\begin{aligned} G &= G_1 \wedge G_2 \\ A &= (G_1 \wedge G_2) \Rightarrow (A_1 \wedge A_2) \end{aligned} \tag{4}$$

It is perhaps more intuitive to view the formula as follows, where the guarantees of each component explicitly affect the assumptions of the other:

$$\begin{aligned} G &= G_1 \wedge G_2 \\ A &= (G_1 \Rightarrow A_2) \wedge (G_2 \Rightarrow A_1) \end{aligned} \tag{5}$$

```
refinement::bscu my_bscu:
    components:
        sub_bscu sub1
        sub_bscu sub2
        switch sw
    connections:
        my_bscu.pedal_1 -- sub1.pedal
        my_bscu.pedal_2 -- sub2.pedal
        my_bscu.cmd -- sw.cmd
        sub1.cmd -- sw.cmd_1
        sub1.error -- sw.error
        sub2.cmd -- sw.cmd_2
```

Fig. 6. Refinement specification for the BSCU

```
my_bscu.pedal_1: 3.0
my_bscu.pedal_2: 5.0
sub1.pedal: 3.0
sub2.pedal: 5.0
sub1.cmd: 4.0
sub1.error: False
sub2.cmd: 6.0
sub2.error: True
sw.cmd_1: 4.0
sw.cmd_2: 6.0
sw.error: True
sw.cmd: 7.0
my_bscu.cmd: 7.0
```

Fig. 7. The counterexample obtained from the refinement check of the `bscu`. Note that `my_bscu.cmd` is 7.0, which does not follow the contract, and that the value of `sw.error` and `sub2.error` are the same.

However Formula 4 is equivalent and allows for a more efficient implementation since it linearly extends to an arbitrary number of components.

The Assume/Guarantee contract operations give incorrect results for variable alphabets [7], that is, multiple "names" for the same variable. In our case, two connected ports would constitute the same variable, since their values are tied, but each port has a different name.

We perform alphabet equalization [7] through a refinement specification, which states which instances exist and how they are connected (see Figure 6). This refinement specification is the same configuration as in Figure 4(a).

Given a specification, the contract refinement check is done as follows:

1) Instantiate a variable of the appropriate type for each port in a contract definition for each instance of a component that follows that contract (e.g. `real sub1.pedal`, `bool sw.error`, etc.)
2) For each connection, add an equality constraint between the ports.
3) Instantiate the assumptions and guarantees in each contract for each instance of a component that follows that contract, replacing each variable with the corresponding instantiated variable from step 1.
4) Apply the composition operation to the instantiated contracts. The resulting assumptions and guarantees serve as the implementation contract for the refinement check.
5) Apply the refinement operation to the top level contract and the composition contract.
6) Negate the resulting predicate and search for a solution. If a solution is found it serves as a counterexample. Not finding a solution to the negation is a proof that the refinement check is applicable for all states in the system.

We use an SMT [26] solver to solve this constraint problem. Propositional logic with linear real arithmetic as in the running example is decidable, meaning in this case the SMT solver can always decide the satisfiability of the refinement problem. More powerful logics and more complex arithmetic require other approaches, which we discuss briefly in section VI.

Given a refinement specification for Figure 4(a), we get a valid refinement check, but given a refinement specification for Figure 4(b), we get the counterexample in Figure 7.

According to that counterexample, the switch picks the wrong input when there is an error (its output value is 7.0, which is the invalid `sw.cmd_1` signal plus 1). Since the `switch` works by picking the second input if there is an error, connecting the same `sub_bscu` to the error port and to `sw.cmd_2` does not work.

C. Contract-Based Design Synthesis

With our Computational Design Synthesis (CDS) approach (Section III-A) we were able to generate 68 architectures for the Wheel Braking System use case. With Contract-Based Design (CBD) (Section III-B), we were able to verify 2 of those architectures. Since both approaches are mechanized, the next step is to integrate the two approaches.

Our CDS tool exports to other tools through a notion of an *asset*. An asset is a special component that maps ports and attributes in the CDS language to ports and attributes of a *library component* in another other tool. A library component can be a standard engineering component (like a spring) included with the tool or a custom model developed in that tool. The mapping of ports and attributes from the CDS component to the custom component ensures they stay consistent.

This exporting mechanism is used to integrate CBD into the synthesis process. Assets are used to map the components in the system model to their respective contracts as in Figure 8. Inside a `mapping` block, the ports on the left side correspond to the ports in the system model component, and the ports on the right side are those of the library component. The port names are coincidentally the same for this example, but port names are not assumed to be the same in the tool, hence the repetition in the mappings.

With the assets in place, we can export and check the refinements for each generated architecture. This works fine as for any other export, but for contracts we do the process more intelligently. As stated in Section III-A, synthesis is achieved through a SAT solver to obtain an assignment to a set of boolean variables representing the possible connections. For the contract refinement check we use an SMT solver which also supports boolean variables among others types. We take advantage of this by translating the assignment from

```
asset::contract bscu:
    info:
        name = contract.bscu
    mapping:
        pedal_1 -> pedal_1
        pedal_2 -> pedal_2
        cmd -> cmd

asset::contract sub_bscu:
    info:
        name = contract.sub_bscu
    mapping:
        pedal -> pedal
        cmd -> cmd
        error -> error

asset::contract switch:
    info:
        name = contract.switch
    mapping:
        cmd_1 -> cmd_1
        cmd_2 -> cmd_2
        error -> error
        cmd -> cmd
```

Fig. 8. Asset mapping between the system model and the contracts.

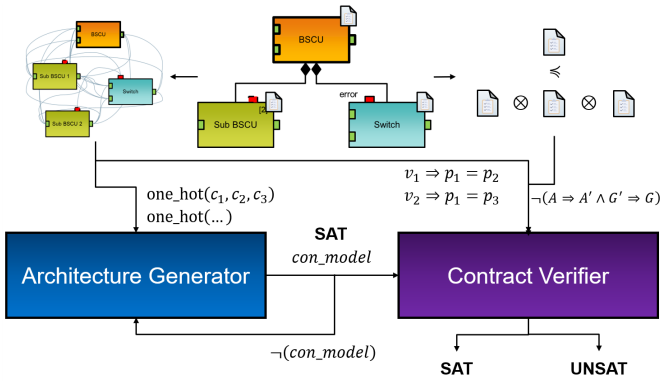


Fig. 9. The architecture of our solver. The Architecture Generator produces configurations from the set of possible connections, and each configuration is then passed to the Contract Verifier for a refinement check.

the SAT solver into an assignment in the SMT solver, where the translation of a refinement check into SMT happens only once. We could have used the SMT solver for both problems, but the SAT solver was an order of magnitude faster in this case. We believe this is due to the high cost of extracting the model from the SMT solver.

The solver architecture can be seen in Figure 9. From the system model, annotated with contract assets, we extract the set of connections on one end and the contract refinement on the other. The connections are translated to `one-hot` constraints and passed to the *Architecture Generator*. The contract refinement is translated to an SMT formula, plus a set of implications from boolean variables to equalities, and passed to the *Contract Verifier*.

As the *Architecture Generator* finds solutions, we extract the assignments to the boolean variables that represent the connections and pass them as assumptions to the *Contract Verifier*. The assignments to the variables trigger the implications which set the equalities in the refinement check. With this encoding we are enable to generate and check all solutions for the running example in 200 milliseconds.

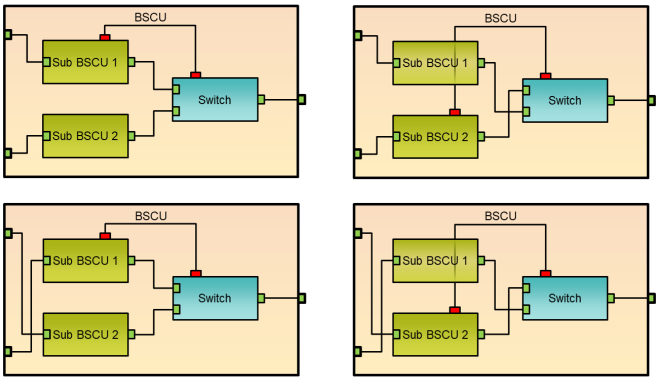


Fig. 10. The 4 valid architectures for the refinement of the `bscu` contract.

With the normal exporter, each configuration is written out to a textual refinement specification, which must then be parsed, interpreted, the contract operations applied, and the refinement check converted to SMT before solving. This is over an order of magnitude slower than the architecture in Figure 9. In the end we get the four solutions in Figure 10.

We also explored an alternative solver which handles everything at the SMT level. Instead of a proof by contradiction, we encoded the refinement check as a universally quantified predicate. Unfortunately, at least for our choice of SMT solver (Z3 [26]), this encoding did not always work. Depending on the seed the SMT solver used internally, we sometimes got fewer than the 4 expected solutions. This could be related to the heuristics used in the solver to handle quantifiers. First-order logic is undecidable in general and only some fragments are decidable [27]. Our encoding and example are in a decidable fragment: the Bernays-Schönfinkel class (i.e. Prenex Normal Form) with linear real arithmetic terms (which admit quantifier elimination). However, in practice, whether the solver terminates depends on the algorithm and heuristics used, i.e pattern-based quantifier instantiation is incomplete [28].

IV. Case Studies

In this section, we will look at two case studies: the Wheel Braking System (WBS) case that served as the running example in this paper, and a more complex Environment Control System (ECS). Some statistics on the two case studies are available in Table II.

A. Wheel Braking System (WBS)

We introduced a simple Wheel Braking System (WBS) use case in Section III where we focused on the synthesis of its Braking System Control Unit (BSCU).

Synthesizing all 68 configurations takes only 3 milliseconds on a 4-core mobile Intel i7-8650U CPU. If we check the refinement of all configurations, the total time stays below

Statistic	WBS	ECS
Components	7	25
Sub–Systems	1	2
Parameters	0	18
Ports	2	337
Assumptions	10	21
Guarantees	6	191
Lines of Code	100	1222

TABLE II
STATISTICS ON EACH OF THE CASE STUDIES.

200 milliseconds. The alternative solver with the quantified encoding mentioned at the end of Section III solves the entire problem in less than 80 milliseconds, though we noticed that depending on the seed of the SMT solver it sometimes produced incorrect results. We discussed possible reasons for the issue in Section III-C.

We also exported the model to Component Fault Trees (CFT) [29] and performed fault tree analysis on each solution. The fault tree analysis cannot take into account the actual behavior of the switch, so it can only detect faults in architectures where the switch is bypassed. It was also and order of magnitude slower, taking 2 seconds to analyze all 68 solutions. Most of that time is overhead since there is interaction over a local socket with an external tool, but this only further validates the need to filter out configurations before exporting to other tools.As stated previously in section III-A, this is not an issue with just 68 possible solutions, but realistic systems can generate millions [16].

B. Environment Control System (ECS)

For a more realistic use case, we modeled an Environment Control System (ECS) for an aircraft. Requirements were collected from Federal Aviation Administration (FAA) [30] and National Research Council (NRC) documents [31], among others, plus boundary conditions obtained from simulation models.

The ECS was first modeled in Simcenter Amesim [12] and then in both Argosim Stimulus [20] and OCRA [32]. Simcenter Amesim was used to run simulations of the ECS, to ensure it had the correct behavior. Stimulus was used for test generation from the contracts. As Stimulus does not directly support Assume/Guarantee contracts, but allows for formal requirement specifications, the contracts were adapted in a straightforward way. Finally the ECS case was modeled in OCRA, which uses bounded model checking to verify contracts with temporal logic constraints.

Both Stimulus and OCRA were able to detect issues with the contract refinement. OCRA found issues in a very short amount of time (a few seconds), but Stimulus was able to find more edge cases [33].

After fixing the issues with the contracts, we modeled the ECS case in our synthesis tool, with all connection constraints removed except for port type constraints. The idea was to export and check the generated configurations on both Stimulus and OCRA and contrast the results. However, neither Stimulus nor OCRA are designed in a way that makes tool interoperability feasible. Stimulus models are purely graphical, with an undocumented JSON-like format for storage, and OCRA only has an interactive command-line mode with a very slow startup time. Because of that we used our contract tool to check a simplified form of the contracts, with temporal constraints removed as CONDEnSe does not yet support them. These were around 14% of the total assertions.

CONDEnSe was able to generate and verify hundreds of configurations per second, but we noticed that many of the architectures that passed the refinement check were not sensible from an engineering point of view, which signaled to us that our contracts were too lenient. In this way, CONDEnSe was used to evaluate the quality of the requirements themselves, in a similar way to mutation testing where a tool slightly modifies the software to be tested to include one or more errors, called *mutants* [22].

The goal of mutation testing is to check the quality of the test suite itself, which should be able to catch most of these *mutants*. By reconfiguring the system in different ways we were, in a way, introducing "mutants" for the contract refinement check. Unlike with mutation testing, it is not possible to automatically detect if the problem is in the configuration or in the requirements, so we rely on human intuition. Since the refinement check was not catching many of the generated "mutants", and those configurations were intuitively broken, this meant the quality of the contracts themselves was low. This cemented our original expectation that computational design synthesis and contract-based design are highly synergistic.

In Section II we identified four weaknesses of computational design synthesis and two weaknesses of contract-based design. The last weakness was left as future work, leaving the following four:

- How to cover more of the design process, particularly requirements?
- How to avoid a combinatorial explosion in the number of variants generated?
- How to safely modularize the process?
- How to ensure the quality of the contracts?

Our solution solves all four challenges. Requirements are covered in the form of contracts, which participate in the synthesis process to filter out bad configurations. Because fewer configurations are returned, it is no longer necessary to manually restrict the model itself, avoiding design fixation. The contract meta-theory guarantees safe modularization, and the synthesis process serves as a measure of the quality of the contracts. This validates the synergy between the two approaches. Previous work showed the scalability of our CDS approach [16], and the modularity of contracts ensures that larger systems can be broken down into manageable chunks. CONDEnSe does not yet support temporal constraints, so it is not yet able to model realistic industrial requirements. How

expressive the tool can be while remaining scalable is an open research question.

V. Related Work

A. Computational Design Synthesis

We introduced the idea of Computational Design Synthesis (CDS) in Section III-A. Our approach is only one form of CDS. Parameter optimization tools like HEEDS [34] are a form of CDS, since they result in multiple design variants when there are multiple key performance indicators (KPIs) that have to be balanced. Our synthesis approach is complementary to parameter optimization, as for each variant generated by CONDEnSe there can be multiple optimal sub-alternatives for different KPIs. Other forms of synthesis include Generative Adversarial Networks (GANs) [35], Genetic Algorithms [36], and Reactive Synthesis [37]. The last two approaches are mainly of interest for controller generation and complementary to our approach, as they would enable us to synthesize correct-by-construction controllers from our contracts, rather than simply assuring the consistency of manually developed controllers. This would increase the amount of automation in our approach. It is worth noting that our approach is only applicable in cases where the system is decomposable.

B. Contract-Based Design

We introduced Contract-Based Design (CBD) in Section III-B. Tools like OCRA [32] and AGREE [38] support contracts with different types of temporal constraints which CONDEnSe does not yet support (support for temporal constraints is ongoing work), but they have no synthesis support. The main inspiration for our approach comes from an article by Nuzzo et al. [39] where they model an electrical power system (EPS) for an aircraft. They are able to synthesize various architectures for the EPS, and synthesize controllers for each of those architectures, but the encoding of the constraints is manual and use case specific.

C. Requirements Engineering

The consistency issues with requirements are a well known problem, and various tools and approaches exist to deal with this issue. As requirements are commonly written in informal language, some tools like Reuse Quality Studio [21] attempt to improve the quality of the text itself in order to reduce the likelihood of consistency problems or misinterpretations. Other tools like Argosim Stimulus [20] focus instead on modeling requirements in a formal way, such that their consistency can be computationally verified. Contracts are a form of the latter, with the added advantage of modularity.

D. Traceability Modeling

Traceability modeling is the typical approach used in industry to solve the issue of consistency management. Tools like Siemens Teamcenter [4] allow the engineer to link each component in the system model to the various artifacts that relate to it, from requirements to simulation models. While traceability tools cannot detect consistency issues, they inform the engineer when a particular artifact is changed, such that all related artifacts can be investigated. While less automated than our approach, it is able to support any kind of model. The issues with manual traceability management are well known in the Software Engineering community, and there are various research efforts to automate it [40]–[42].

E. Product Line Engineering

Product Line Engineering (PLE) is an approach to systematically manage variation across a family of products [43]. While our approach focuses on variation at the architectural level (meaning configurations of components), PLE works at the level of features, which is a higher level view (a feature is a characteristic or end-user-visible behavior of a system). In order to achieve that, we need to have a system in place first. That is where our methodology comes in. Since PLE works on another level from our approach in the design process, it is orthogonal but complementary to our method. An open problem, for example, is the detection of features in generated architectures, i.e., is a specific architecture a series hybrid configuration? For software, PLE is a better fit than our approach, as features can be mapped directly to conditional compilation points, whereas what constitutes a component or a connection between components in a software system is less well defined.

VI. Conclusions and Future Work

In this paper, we pointed out the difficulty of maintaining consistency between various artifacts produced throughout the design process, and how Computational Design Synthesis (CDS) and Contract-Based Design (CBD) attempt to solve this challenge.

We recognized gaps in both approaches and we demonstrated that they can be reduced when CDS and CBD are applied in a synergistic way.

To illustrate our approach, we introduced a methodology and tool that maintains consistency between formal requirements, the system model, design variants, and various simulation and analysis models. The tool has a high degree of automation and supports modular design.

Future work for our research is to understand how to support a wider variety of requirement specifications, particularly realistic industrial requirements with temporal constraints. This may require using multiple solvers for different kinds of specifications, which in turn leads to performance challenges related to the higher complexity of requirements.

Acknowledgment

This research was supported by the Belgian funding agency VLAIO. We thank our colleagues from Siemens who provided insight and expertise that assisted the research, and we also thank our colleagues from KU Leuven and the three anonymous reviewers for their comments that greatly improved the manuscript.

REFERENCES

[1] R. Bishop, *The Mechatronics Handbook*. Jan. 2002.

[2] J. Torry-Smith, A. Qamar, S. Achiche, J. Wikander, N. H. Mortensen, and C. During, "Challenges in Designing Mechatronic Systems," *Journal of Mechanical Design*, vol. 135, Jan. 2013.

[3] H. Van der Auweraer, J. Anthonis, S. De Bruyne, and J. Leuridan, "Virtual Engineering at Work: the Challenges for Designing Mechatronic Products," *Engineering with Computers*, vol. 29, pp. 389–408, Sept. 2012.

[4] "Siemens Teamcenter." https://www.plm.automation.siemens.com/global/en/products/teamcenter/. Accessed: 2019-04-26.

[5] P. Micouin, *Model-based systems engineering: fundamentals and methods*. Control, systems and industrial engineering series, Hoboken, NJ: Wiley, 2014. OCLC: 1015344478.

[6] L. Delligatti, R. Steiner, and R. Soley, *SysML distilled: a brief guide to the systems modeling language*. Upper Saddle River, NJ: Addison-Wesley, 2014. OCLC: 873412584.

[7] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. L. Sangiovanni-Vincentelli, W. Damm, T. Hezinger, and K. Larsen, "Contracts for System Design," Research 8147, INRIA, 2012.

[8] J. Cagan, M. I. Campbell, S. Finger, and T. Tomiyama, "A Framework for Computational Design Synthesis: Model and Applications," *Journal of Computing and Information Science in Engineering*, vol. 5, no. 3, p. 171, 2005.

[9] D. Lanzarotto, M. Marchesoni, M. Passalacqua, A. P. Prato, and M. Repetto, "Overview of different hybrid vehicle architectures," *IFAC-PapersOnLine*, vol. 51, no. 9, pp. 218 – 222, 2018.

[10] E. Ruijters and M. Stoelinga, "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools," *Computer Science Review*, vol. 15-16, pp. 29–62, Feb. 2015.

[11] J. Allen and P. Kouppas, "Computer Aided Design: Past, Present, Future," pp. 97–111, Jan. 2012.

[12] "Simcenter Amesim." https://www.plm.automation.siemens.com/global/en/products/simcenter/simcenter-amesim.html. Accessed: 2019-04-26.

[13] "Simcenter 3D." https://www.plm.automation.siemens.com/global/en/products/simcenter/simcenter-3d.html. Accessed: 2019-04-26.

[14] A. Madni and S. Purohit, "Economic Analysis of Model-Based Systems Engineering," *Systems*, vol. 7, p. 12, Feb. 2019.

[15] M. Nicolai, L. Salemio, and J. Vanhuyse, "Design Space Modeling Language for the Generation of Engineering Designs." Pub. No. US 2017/0316126 A1, Filed May 2, 2016.

[16] H. Karhula, M. Nicolai, and W. Desmet, "Exhaustive Synthesis And Analysis Of Automotive 2-Stage Planetary Transmission Designs," in *Proceedings of the ASME 2019 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, ASME.

[17] J. Vanhuyse, N. Murgovski, M. Nicolai, H. Van der Auweraer, and W. Desmet, "Closed-Loop Design for Hybrid Powertrains," in *Proceedings of the ASME 2017 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, p. V003T01A024, ASME, Aug. 2017.

[18] J. Menu, M. Nicolai, and M. Zeller, "Designing Fail-Safe Architectures for Aircraft Electrical Power Systems," American Institute of Aeronautics and Astronautics, July 2018.

[19] D. Firesmith, "Common requirements problems, their negative consequences, and the industry best practices to help solve them.," *Journal of Object Technology*, vol. 6, pp. 17–33, 01 2007.

[20] "Argosim Stimulus." https://www.argosim.com/home/stimulus-for-requirements/. Accessed: 2019-04-26.

[21] "Reuse Quality Studio." https://www.reusecompany.com/rqa-quality-studio/. Accessed: 2019-04-26.

[22] P. Reales Mateo, M. Polo, J. Fernndez-Alemn, A. Toval, and M. Piattini, "Mutation Testing," *Software, IEEE*, vol. 31, pp. 30–35, May 2014.

[23] A. Cimatti and S. Tonetta, "A Property-Based Proof System for Contract-Based Design," pp. 21–28, IEEE, Sept. 2012.

[24] P. Gu, J. Purdom, J. Franco, and B. Wah, "Algorithms for the Satisfiability SAT Problem: A Survey," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pp. 19–152, Jan. 2002.

[25] A. M. Frisch and P. A. Giannaros, "SAT Encodings of the At-Most-k Constraint Some Old , Some New , Some Fast , Some Slow," 2010.

[26] L. de Moura and N. Bjrner, "Z3: an efficient SMT solver," vol. 4963, pp. 337–340, Apr. 2008.

[27] P. Fontaine, "Combinations of Theories for Decidable Fragments of First-Order Logic," vol. 5749, pp. 263–278, Sept. 2009.

[28] L. M. d. Moura and N. Bjrner, "Efficient E-Matching for SMT Solvers," in *CADE*, 2007.

[29] M. Zeller, *Safety and Reliability of Complex Engineered Systems: ESREL 2015*. CRC Press, Sept. 2015.

[30] "FAA: Federal Aviation Administration. Part 25 airworthiness standards: Transport category airplanes." https://www.ecfr.gov/cgi-bin/text-idx?node=14:1.0.1.3.11. Accessed: 2019-04-26.

[31] N. R. Council, ed., *The airliner cabin environment and the health of passengers and crew*. Washington, DC: National Acad. Press, 2002. OCLC: 248643824.

[32] A. Cimatti, M. Dorigatti, and S. Tonetta, "OCRA: A tool for checking the refinement of temporal contracts," pp. 702–705, IEEE, Nov. 2013.

[33] L. Vos, "Contract-based design of environmental control systems for aircraft," Master's thesis, KU Leuven, 2018.

[34] "Simcenter HEEDS." https://www.plm.automation.siemens.com/global/en/products/simcenter/simcenter-heeds.html. Accessed: 2019-04-26.

[35] K. Wang, C. Gou, Y. Duan, L. Yilun, X. Zheng, and F.-Y. Wang, "Generative Adversarial Networks: Introduction and Outlook," *ACTA AUTOMATICA SINICA*, vol. 4, pp. 588–598, Nov. 2017.

[36] P. Fleming and R. Purshouse, "Genetic Algorithms In Control Systems Engineering," May 2002.

[37] V. Raman, A. Donz, D. Sadigh, R. M. Murray, and S. A. Seshia, "Reactive Synthesis from Signal Temporal Logic Specifications," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, (New York, NY, USA), pp. 239–248, ACM, 2015. event-place: Seattle, Washington.

[38] D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional Verification of Architectural Models," in *NASA Formal Methods*, 2012.

[39] P. Nuzzo, Huan Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donze, and S. A. Seshia, "A Contract-Based Methodology for Aircraft Electric Power System Design," *IEEE Access*, vol. 2, pp. 1–25, 2014.

[40] A. Egyed, "A Scenario-driven Approach to Traceability," in *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, (Washington, DC, USA), pp. 123–132, IEEE Computer Society, 2001. event-place: Toronto, Ontario, Canada.

[41] J. Adersberger and M. Philippsen, "ReflexML: UML-Based Architecture-to-Code Traceability and Consistency Checking," in *Software Architecture* (I. Crnkovic, V. Gruhn, and M. Book, eds.), Lecture Notes in Computer Science, pp. 344–359, Springer Berlin Heidelberg, 2011.

[42] J. Thangarajah, G. Jayatilleke, and L. Padgham, "Scenarios for System Requirements Traceability and Testing," in *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '11, (Richland, SC), pp. 285–292, International Foundation for Autonomous Agents and Multiagent Systems, 2011. event-place: Taipei, Taiwan.

[43] K. Pohl, G. Bckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Heidelberg: Springer-Verlag, 2005.